

U.S. PATENT APPLICATION  
FOR  
A MECHANISM FOR SELECTING REPRESENTATIVES  
FROM PROGRAM PATCH CHAINS  
BASED ON USER ROLES

Inventors: Evan R. Zweifel  
305 East Magnolia Street  
Fort Collins, CO 80524

Brent A. Allsop  
4034 Stoneway Court  
Fort Collins, CO 80525

Joseph M. Webster  
2930 Worthington Avenue  
Fort Collins, CO 80526

# A MECHANISM FOR SELECTING REPRESENTATIVES FROM PROGRAM PATCH CHAINS BASED ON USER ROLES

## CROSS REFERENCE TO RELATED APPLICATION

[0001] This application is a continuation-in-part of application serial No. 09/924,773 filed on August 8, 2001, which application is hereby incorporated by reference into the present application, including its appendix (which is not reproduced below).

## BACKGROUND OF THE INVENTION

### Field of the Invention

[0002] The present invention relates generally to techniques for maintaining programming systems, and more particularly, to methods for selecting which sets of program corrections or “patches” are to be installed in accordance with the security needs of a particular organization.

### Description of the Related Art

[0003] When programs are installed upon a computer system, the programs are constituted of a large number of individual files which are grouped together into what may be called “filesets.” For example, in Fig. 2 at 200, a systems database is shown which lists the names of various systems (SYSTEM A, SYSTEM B, etc.) and then lists following each system’s name the “filesets” that are installed upon that system and the files which each of those “filesets” contain. For example, the SYSTEM A contains the FILESETS FS1 and FS2. The FILESET FS1 is shown in Fig. 2 as containing the files FILE A, FILE B,..., and FILE F. Likewise, the FILESET FS2 is shown as containing the files FILE J, FILE K,..., and FILE P.

[0004] As time passes, both through the detection of defects in the various files and also through changes in the needs of the users of the system, corrections and improvements are made to the files that comprise a given system. These are distributed in the form of “patches” each of which contains a number of files that are basically updates and improvements to the files previously installed. It is

customary to group all the files contained within a given patch into one or more filesets, and to give the filesets within a patch the same names as the filesets to which they correspond in the actual systems. Accordingly, and with reference to Fig. 3 at 300, a PATCHES DATABASE is shown. A first patch, named PATCH\_5, contains a fileset named FILESET FS1 which contains only an updated copy of the single file FILE A. In practice, all computer systems containing FILESET FS1 would be updated with PATCH\_5. The updating process replaces a copy of the FILE A originally installed on the system with the newly revised copy of FILE A that is contained within the patch.

[0005] Over time, further patches are issued for a given system. In Fig. 3, an additional patch, PATCH\_8 contains updates for FILESET FS2 which, in this case, constitutes the single updated file FILE K. At a later time, an even newer patch, PATCH\_6 issues which contains updates for both the file sets FILESET FS1 and FILESET FS2. Of necessity, the patch PATCH\_6 coming later in time than the other two patches, PATCH\_6 includes all the updates of the earlier patches plus some new updates. More specifically, the patch PATCH\_6 includes a set of updated files named FILESET FS1 that replace both FILE A and FILE F, as well as another set named FILESET FS2 which contains replacement copies of FILE K and FILE P. As is apparent, if the SYSTEM A had not previously been updated with the patches PATCH\_5 and PATCH\_8, that system could be fully updated with the single patch PATCH\_6 and would not need updating with the earlier patches. In that sense, the patch PATCH\_6 SUPERSEDES and replaces the earlier patches, which may be called “predecessors” of PATCH\_6. In the discussion which follows, a “predecessor” patch is sometimes called a “child” patch, and “predecessors” are sometimes called “children.”

[0006] Fig. 4 at 400 presents a patch tree database which illustrates a way in which the historical and shared file relationships between patches can be represented in a searchable database. In Fig. 4, the newest patch PATCH\_6 is shown in a central column that is labeled ROOT PATCHES. Extending to the left from this newest patch is a patch tree structure, which in this case contains only the two patches PATCH\_5 and PATCH\_8 shown as two limbs of a tree that converges upon the root PATCH\_6. The tree portion of Fig. 4 is labeled TREE PATCHES to

distinguish it from the ROOT PATCHES portion which contains the root of the patch trees. To the left in Fig. 4 is a column labeled FILESETS which simply lists all the filesets that are contained within the root patches of the patch trees. While only one patch tree is shown in the patch tree database 400, typically such a database would contain numerous trees each having a root patch and each relating to a number of different filesets. For example, the patch trees shown in FIG. 15 at 1500 could occupy a common patch tree database 400.

[0007] Figs. 4 and 15 also illustrate a number in parenthesis opposite the name of each patch. This number indicates the reliability of each patch. A rating of "1" indicates that a patch is new and has undergone little testing. A rating of "2" indicates that the patch has been available for use for some limited amount of time and has been installed on at least some minimal number of systems. A rating of "3" indicates that the patch has undergone some system testing. Clearly, a higher rated patch corresponds to a more tested patch and therefore a more reliable patch.

[0008] In the past, it has been customary any time a system is updated to install only the newest set of root patches that contain filesets corresponding to the filesets installed on a given system. In this manner, a system is kept up-to-date. However, some of the patches installed may not have undergone sufficient testing to suit the needs of a system that is mission critical and that should not be updated with patches until they have undergone fairly thorough testing. A trained technical expert can go through all the patches, looking at the date of each patch and estimating its reliability, and can then select patches which have been around for sufficient time so that their reliability is fairly certain. However, this is a time consuming process that can also result in erroneous selections.

[0009] Some patches may also need to be made invisible to all save certain individuals, such as programmers and beta testers. Some patches may need to be confined to "in-house" technicians or may need to be taken out of service entirely for various reasons.

#### SUMMARY OF THE INVENTION

[00010] Briefly summarized, an embodiment of the invention may be found in a method of selecting program patches for installation by human or machine patch

recipients into computer systems, where the patches are organized into patch chains each having a root. The method comprises the steps of searching for a patch that corrects a particular defect or that has a particular property or both; examining additional patches, if any, sharing the same patch chain as any such patch found as a result of the search and occupying a position on the shared patch chain between that of any such patch found and the root of the patch chain; and presenting one or more patches, including any such patch found and examined patches that satisfy one or more specified conditions determined by the nature of each patch and the identity of the patch recipient.

### BRIEF DESCRIPTION OF DRAWINGS

[00011] Fig. 1 presents an overview block diagram of the patch selection method of the present invention.

[00012] Fig. 2 presents the structure of a systems database that indicates which files, which filesets, and which patches are installed on each system.

[00013] Fig. 3 presents the structure of a patches database that indicates what filesets each patch corrects and which files within those filesets the patches repair or modify or both.

[00014] Fig. 4 presents the database structure of a patch tree database showing the root patch for each patch tree, the filesets that each patch tree modifies, and the non-root patches within the branches of each patch tree.

[00015] Fig. 5 presents a flow diagram of a function which, given a list of the names of patches already installed on a system and a list of the names of the root tree patches for the patch trees that contain modifications to the filesets of the system, returns a list of recommended new patches for the system in the form of triples.

[00016] Fig. 6 presents a flow diagram of a recursive function that is called by the function shown in Fig. 5 to trace recursively through the individual patch trees and sub-trees searching one tree or sub-tree during each recursion, to find recommended patches for system update.

[00017] Fig. 7 is a continuation of the flow diagram of Fig. 6.

[00018] Fig. 8 is a continuation of the flow diagram of Figs. 6 and 7.

[00019] Fig. 9 is a subroutine that determines whether the patch at the root of a given sub-tree is a better choice than at least one of the patches in that sub-tree's branches.

[00020] Fig. 10 presents a simple linear patch tree.

[00021] Fig. 11 presents a more complex patch tree with several branches.

[00022] Fig. 12 presents a set of four patch trees, two of which have branches.

[00023] Fig. 13 illustrates a patch tree in which the patches have ratings assigned to them.

[00024] Fig. 14 illustrates the patch tree shown in Fig. 13 at a later time, with a new root patch and with the ratings updated to reflect a new patch and further usage and testing.

[00025] Fig. 15 presents an illustrative set of patch trees.

[00026] Fig. 16 illustrates a possible set of root patch names and installed patch names which, when analyzed in accordance with the function shown in Fig. 5, produces the resultant set of patch installation recommendation triples also shown in Fig. 16.

[00027] Fig. 17, presented in flow diagram form, presents, as an alternative embodiment of the invention, a method for selecting a set of patches to a particular user that address a particular problem of the user, that match the user's need for reliability, and that are available to the particular user, using a forward search approach that searches a patch tree from a first identified patch forward to the root of the tree.

[00028] Fig. 18 is a continuation of Fig. 17, plus an illustrative presentation of a set of patches to a user or recipient in tabular format.

## DETAILED DESCRIPTION

[00029] (The particular embodiment of the invention summarized above and claimed in the appended set of claims is described most fully in Figs. 17 and 18 and in the description of Figs. 17 and 18 that is presented below.)

[00030] As an aid to understanding the present invention, Figs. 10-14 present simple examples of patch tree data structures that are described in the following paragraphs.

[00031] When Hewlett-Packard's version of UNIX "HP-UX," receives new program files that are to be added to a given system, the files are delivered gathered into filesets having names, such as FS1, FS2, and so on. These filesets are installed upon a given system by a process that unpacks and, possibly, uncompresses the files and places them onto the hard disk drive of that system. As in shown in Fig. 2, each fileset can contain a small or large number of files. The FILESET FS1 is shown containing the files FILE A, FILE B, ...and FILE F. Likewise, the FILESET FS2 is shown containing the files FILE J, FILE K, ...and FILE P. Of course, a fileset typically contains many more files than this. Some of these would be program files, some would be data files, some would be graphic image and multimedia files, depending upon the particular nature of the system and the particular nature of the programming system being installed.

[00032] Patches, or corrected/updated sets of files, are also delivered to a system as collections of filesets within each patch. In the HP-UX system, it is customary that the filesets in a patch have the same names as the installed filesets. A patch fileset will contain updated versions of some (possibly all) of the files in the system fileset having the same name. A given patch PATCH\_5 contains new features and fixes or repairs for specific defects. Descriptions of the new features and of the repaired defects are contained in a text file that this maintained in a central database for each patch and that is searchable for words and phrases. Accordingly, a systems administrator may search through the patch text file database and locate patches that repair particular defects or add particular features.

[00033] Over time, a first patch may be replaced by a second patch which contains all the fixes and new features of the first patch plus additional changes. These additional changes are called incremental fixes. The new patch then SUPERSEDES the previous patch. With reference to Fig. 10, the PATCH\_4 at the root of the patch tree 1000 supersedes all of the three patches to the left in this simple linear search tree. Historically, the first patch created was PATCH\_1. It was superceded by PATCH\_2, which was later superceded by PATCH\_3, and that

patch was later superseded by PATCH\_4 which now resides at the root of the patch tree 1000.

[00034] In some situations, as illustrated in Fig. 11 at 1100 and also in Fig. 4 at 400, two or more patches will be replaced by a single patch. Thus, PATCH\_6 SUPERSEDES both the patches PATCH\_5 and PATCH\_8. This is represented in the search tree by PATCH\_6 forming the root of a sub-tree having the two branches PATCH\_5 and PATCH\_8. Referring now to Fig. 11, the same patch tree shown in Fig. 4 is shown at a later point in time. At some point in time, a new patch PATCH\_9 was added which was not part of the original patch search tree but which initially formed a single isolated patch search tree having only one patch element. Then a new patch PATCH\_7 was created which combined all of the updates and changes contained in the patches 5, 6, 8, and 9. Even later on, that patch was superseded by a new patch PATCH\_10, thus forming the patch tree 1100 shown in Fig. 11. The root patch in the patch tree 1100 is the PATCH\_10. That patch and PATCH\_7 form the trunk of this searchable patch tree, which then branches into two branches, one containing PATCH\_9 and another containing PATCH\_6; and the PATCH\_6 branch of the tree then branches again into the two patches PATCH\_5 and PATCH\_8. As can be seen, a patch tree can become quite elaborate over time as many patches are combined into a smaller number of newer patches. When placed into a patch tree database, as shown in Fig. 4, a patch tree can be searched in an automated manner, as will be explained.

[00035] Typically, large systems will contain large numbers of filesets, and these will be updated by the patches in multiple disjoint patch trees (i.e. a patch will appear in at most one tree). Accordingly, Fig. 12 illustrates a possible set of four patch trees 1202, 1204, 1206, and 1208 all comprising a set of patches 1200 that are used to update a given system. The set of patch trees shown in Fig. 12 is selected by first determining what filesets a given system contains and by then, with reference to a patch tree database such as that shown at 400 in Fig. 4, selecting the root patches for all the patch trees that contain filesets having the same names as the system filesets.

[00036] The beginning point for the patch selection method of the present invention is the determination, at steps 104 and 106 in Fig. 1, of the names of all



the root patches that contain filesets whose names correspond to the names of a given system's filesets. These fileset names are first retrieved from a systems database 200 (Fig. 2), and the same fileset names are then located in the fileset column of a patch tree database 400 (Fig. 4). The names of the root patches for the corresponding patch trees are then obtained from the root patches column of the patch tree database 400 shown in Fig. 4. The root patch names are then combined as a set and are stored together as a set variable named ROOTS. The set variable ROOTS is adjusted to contain, as set elements, the names of the root patches (PATCH\_6, for example) which the patch tree database 400 links to the fileset names, such as FILESET FS1 and FILESET FS2, that are also the names of the filesets for a given system. Alternatively, file names could be used instead of fileset names for this purpose.

[00037] The patch tree database 400 can be constructed from a patches database 300 (Fig. 3) that shows what fileset names and what files each patch contains, as well as the creation date for each patch. This database 300 can be generated from the uncompressed patches themselves in an automated fashion, if desired.

[00038] The second step needed at the start of the patch selection method of the present invention is to determine which update patches a given system has already received. With reference to Fig. 2, a system's database 200 contains a record of the patches that have already been installed on each given system. This database can be derived from the log files that are generated when a system receives new patches. Thus, the SYSTEM A is shown as having already received the patches PATCH\_5 and PATCH\_8. This corresponds to the step 102 shown in Fig. 1. As indicated at 102 in Fig. 1, the names of these installed patches are combined and are stored within a set variable named INSTALLED, such that each element associated with this variable is the name of a patch already installed on the SYSTEM A.

[00039] In the example illustrated by Figs. 2 and 4, the SYSTEM A includes the two filesets FILESET FS1 and FILESET FS2 both of which filesets, according to Fig. 4, are modified by the patches in the patch tree whose root element is PATCH\_6. Accordingly, in this case the system variable ROOTS is assigned the

single name PATCH\_6 and thus contains the name of only one patch tree. In general, as is illustrated in Fig. 12, several patch trees may be relevant to updating the filesets of a given system. Thus, if the SYSTEM B listed in Fig. 2 contains filesets whose names the patch tree database 400 associates with the set of patch trees shown in Fig. 12, then in that instance the system variable ROOTS will be assigned the four patch tree root patch name values PATCH\_4, PATCH\_10, PATCH\_11, and PATCH\_13 all of which names are retrieved from the root patches column of the patch tree database 400 in Fig. 4.

[00040] Having found the names of all the patches previously installed in a given system, and having associated those names with the system variable INSTALLED; and having also found all of the patch tree root patch names relevant to the updating of a given system, and having associated those names with the system variable INSTALLED; the present invention now passes the two sets of values INSTALLED and ROOTS to a function entitled FIND\_ALL\_I\_R\_L (find all set of the triple values (I, R, L) for this system). As shown at step 108 in Fig. 1, this function returns a set of triple (I, R, L) values.

[00041] Each triple value returned is a recommendation of a possible way to update the system. Within each triple, the central value "R" is the name of a "recommended" patch to be installed on the system, or "R" is NULL if this triple contains no recommendation. This recommended patch name was retrieved from a patch tree. "L" is the name of the root (or "latest" or most recent) patch in that patch tree. "I" in each triple is the name of an already installed patch that is to be superceded by the recommended patch, or else it is NULL if there was no prior patch installed that is being superceded.

[00042] A conservative user will take the name values R, obtain the correspondingly named patches, and install them to update a given system. A user who is not concerned about risks and wants to receive the very latest updates can, instead, take the name values L and install them upon a given system. A very conservative user, after taking the name values R, might then obtain the text files describing the recommended patches R and review what those patches do, and then select only those recommended patches containing changes that are important to that

particular user, thereby avoiding the possibility of introducing new problems along with new patches in areas that are irrelevant to a particular user's needs.

[00043] The call to the function FIND\_ALL\_I\_R\_L performed at step 500 in Fig. 5 is drawn to indicate that that function 500 calls a second function FIND\_I\_R\_L 600 to search each individual patch tree, and the function 600 recursively calls itself as needed to examine each patch within each tree. By "recursive," it is meant that this latter function 600 calls upon itself one or more times in the course of searching right-to-left through complex patch trees, examining earlier patches, and determining whether they should be superceded by later patches or whether, due to the low ratings of the later patches, the earlier patches should be utilized instead or retained.

[00044] A user with a particular system is looking for patches that will bring their system up-to-date. With the possibility of different patch ratings for different patches on the same patch tree, the problem arises as to which patch is the most appropriate to be recommended to a given user. The recommendation depends on the amount of risk that a particular user is willing to accept.

[00045] The patch selection algorithm, presented in overview in Figs. 5-9 and in detail in the Appendix to this application, creates a set of recommended patches for the user given a particular patch search space of patch trees and a given description of the patches already installed on a user's system. The recommended patches typically have higher ratings, and thus they introduce minimal additional risk to the user. The recommended patches are represented as sets of triple I, R, L values, as was just explained. The following definition forms the basis for determining whether users should install a given patch R on top of an already installed patch I. This definition is conservative – selecting a successor patch only if it is highly tested, or if it is at least more tested than a currently-installed patch.

[00046] Consider two patches I and R, where R is a successor to I. R is considered "clearly better" than I if and only if:

[00047] The rating of R is greater than the rating of I, or

[00048] The rating of R is 3.

[00049] Consider the exemplary patch tree shown in Fig. 14. In this example, the following conclusions may be drawn:

[00050] PATCH\_6 is clearly better than PATCH\_5 but not PATCH\_8.

[00051] PATCH\_7 is clearly better than PATCH\_9.

[00052] PATCH\_10 is clearly better than PATCH\_5 but not PATCH\_6.

[00053] The following definition makes a patch recommendation from all of the “clearly better” patches. The definition will only recommend less risky patches by selecting patches with a rating of at least 2. The most recent, highest rated patches are selected. Note that the definition still applies when the patch tree contains no installed patches.

[00054] Definition of “recommended.” A patch R is recommend if and only if:

[00055] 1. R has a rating of at least 2.

[00056] 2. There are no successors to R with higher or equal ratings.

[00057] 3. There are no successors to R which are already installed.

[00058] 4. If R is a successor to some set of installed patches, then R is “clearly better” than at least one of them.

[00059] Consider the example set forth in Fig. 13.

[00060] If no patches are installed, the recommended patches are PATCH\_8 and PATCH\_9.

[00061] If PATCH\_5 and PATCH\_9 are installed, PATCH\_7 is recommended.

[00062] The present invention is implemented by means of a program 500 (Fig. 5) named FIND\_ALL\_INSTALLED\_RECOMMENDED\_LATEST or, as depicted in the drawings, FIND\_ALL\_I\_R\_L. This program 500 is implemented as a function returning sets of triples or triple values. A brief explanation of the returned sets of triple values is presented at step 108 in Fig. 1, and was explained above. The calling parameters passed into this function are explained at 502 in Fig. 5. The assembly of these calling parameters is illustrated in Fig. 1 in the steps 102,

104, and 106 which lead up to calling this function at step 500, which steps were explained above.

[00063] As illustrated in Fig. 1, the function FIND\_ALL\_I\_R\_L 500 works by recursively calling a secondary recursive function FIND\_I\_R\_L 600 that is shown in the Figs. 6-8 (with the entry point being the step 602 in Fig. 6) and which calls upon a subroutine 900 that is shown in Fig. 9. A complete pseudo-code listing of all of these programs is presented in the Appendix of this application. The functions presented in the Appendix are fully described and explained by the flow diagrams presented in the Figs. 5-9.

[00064] Figs. 15 and 16 illustrate the use of the invention to select patches from a set of patch trees 1500 that are shown in Fig. 15. Five patch trees 1502, 1504, 1506, 1508, and 1510 are shown in Fig. 15. Each patch tree is identified by the name of the root, or most recent, patch, which appears to the right in Fig. 15. Thus, the patch tree 1502 is identified by the patch name PATCH\_4, the patch tree 1504 is identified by the patch name PATCH\_11, and so on.

[00065] In this example, the set variable INSTALLED, shown at 1604 in Fig. 16, contains the names of all the patches that have already been installed on a hypothetical system. The set variable ROOTS shown at 1602 contains the names of the root patches of the five patch trees 1500 shown in Fig. 15. These values are gathered by performing the steps 102, 104, and 106 shown in Fig. 1, as has been explained. After execution of the function at 500, which calls the recursive function 600, the results of the patch analysis are returned (step 108 in Fig. 1) as a set of six triple values which are shown collectively at 1606 in Fig. 16 to include the individual triple values 1608, 1610, 1612, 1614, 1616, and 1618. By considering the above rules, and by examining the tree structures shown in Fig. 15, as well as the set variables ROOTS 1602 and INSTALLED 1604, it can be seen how these triple values were produced.

[00066] Briefly summarized, the triple 1608 recommends that PATCH\_1 be replaced by PATCH\_3. In the patch tree 1502, PATCH\_3 which is newer and more reliable than PATCH\_1; while the PATCH\_4 is still newer, it is not recommended because of its unreliability.

[00067] The triple 1610 similarly recommends the installation of the new PATCH\_10 to replace the PATCH\_5, but it does not recommend installation of the still newer but unreliable PATCH\_11. The similar triple 1612 recommends that the same PATCH\_10 also replace the previously installed PATCH\_9, even though PATCH\_10 is less reliable than PATCH\_9, since PATCH\_10 has already been recommended to replace the even less reliable PATCH\_5.

[00068] A triple 1614, which relates to the patch tree 1506, does not recommend that the newest PATCH\_13 replace the previously installed PATCH\_14 because they both have a reliability rating of 2 and therefore PATCH\_13 is not “clearly better” than PATCH\_14. This triple 1614 contains a recommendation of NULL.

[00069] The triple 1616 suggests that the PATCH\_15, with a rating of 2, be installed. The NULL value in this triple indicates that no previous patch has been installed.

[00070] The triple 1618 recommends against installing the single PATCH\_16, since it has an unacceptable reliability rating of 1.

[00071] As can be seen in the set of triples shown at 1606 in Fig. 16, the first value of each triple, identified by the letter I, is either a NULL value, or it is the name of a patch that was previously “installed” and that is now being replaced by whatever recommendation is made. The middle value, assigned the letter R, is NULL if no recommendation is being made for a replacement, or it is the name of a “recommended” replacement patch. The third value, identified by L, is the name of the “latest” patch – the one most recently added to the patch tree that contains both the patches I and R. If that latest patch is rated highly and is reliable, it is the choice in every case. That last patch is bypassed simply to give better system stability and reliability at the sacrifice of new features that might have been added by the latest patch. The field engineer, after viewing the text file describing the features that may have been added to the patches, may choose to override the recommendations and go with the latest patch, the one that appears to the right in the patch tree and in each triple, depending upon the needs of a particular system.

[00072] Fig. 5 presents a block diagram description of the function 500 named FIND\_ALL\_I\_R\_L, which is an abbreviation for the function name

FIND\_ALL\_INSTALLED\_RECOMMENDED\_LATEST that appears in the Appendix. Given a set of patch trees (Figs. 4, 12, or 15) relevant to a given system and given a list of the names of the patches already installed on that system, this function produces a series “triples” of recommended patch updates each of which includes the name L of the “latest” patch in a patch tree set, the name R of a recommended patch, and the name I of an installed patch that is to be superceded. The above paragraphs have described the triples 1606 (Fig. 16) returned in a given exemplary situation.

[00073] With reference to Fig. 5, the first step 502 simply describes the incoming arguments passed to this function by the calling program 100 which appears in Fig. 1 and was discussed above. The set variable INSTALLED contains the names of the patches that have already been installed in the system that is to be upgraded. The set variable ROOTS contains the names of the relevant root patches of the patch trees that contain patches relevant to this system’s filesets, as was explained above.

[00074] The function 500 begins at step 504 by setting the set variable TRIPLES equal to NULL. This variable TRIPLES is the return argument which, at step 510, returns the recommendations, as described at 108 in Fig. 1 and as illustrated at 1608-1616 in Fig. 16, to the calling program 100 in Fig. 1.

[00075] Beginning at step 506, this function 500 begins to loop through the steps 506, 600, and 508. Each time through this loop, a temporary variable R is set to the name of one of the patch tree root patch names that is retrieved from the set variable ROOTS. Each time through this loop, the re-enterable function FIND\_I\_R\_L 600 is called and is passed, as the first two of its three incoming arguments, two copies of this variable R which contains the name of the root patch in a patch tree. The third incoming argument is the variable INSTALLED which contains the names of all the installed patches.

[00076] At step 508, any triple values returned by a given call to the function 500 are added to the variable set TRIPLES and are thus preserved to be returned by the function 500 to the calling program 100 when the function 500 terminates execution at step 510. Accordingly, each relevant patch tree is analyzed independently by a call to the function 600, the details of which appear in Figs. 6-9.

That function 600 begins at the root of a patch tree and, by means of recursive calls to itself, moves up the patch tree one step at a time, evaluating every patch in the tree one patch at a time, each patch being evaluated by a separate recursive call to the same function.

[00077] Referring now to Fig. 6, the recursive function FIND\_I\_R\_L 600 begins at step 602 in Fig. 6, where its incoming arguments are described.

[00078] Referring now to Fig. 6, the recursive function 600 has a set of three arguments passed to it, as is indicated at 602. It returns a set of triples, as indicated at 108 in Fig. 1. The incoming three arguments described at 602 include a first argument that is the name of a patch and that changes with each recursive call, and second and third arguments that never change throughout the recursive operation of the function 600, although each time the function 600 is called by the function 500, the second argument, a patch name, changes. The third argument, the set of the names of installed patches INSTALLED, remains invariant at all times.

[00079] The second argument, which is different for each call to the function 600 by the function 500 but which is invariant within recursive calls of the function 600 to itself, is the name of the patch that appears at the root of the particular patch tree that is being evaluated by the function 500 at the request of the function 600. It will be recalled that the function 500 receives these root patch names in the set variable ROOTS. The function 500 calls the function 600 repeatedly, each time varying the root patch tree name that is passed to the function 600 so that a different patch tree is evaluated by each call to the function 600.

[00080] The first argument, CURRENT, is the one that varies with each recursive call of the function 500. Assume, for example, that the function 500, at step 600, is calling upon the recursive function 600 to evaluate the patch tree 1504 shown in Fig. 15. The initial call of the function 500 to the recursive function 600 will set both the value ROOT and the value CURRENT to the name of the that patch tree 1504's root patch, PATCH\_11. Thus, the function 600, before it begins to call itself recursively, is asked to evaluate the CURRENT patch name PATCH\_11 in the patch tree having the root patch name PATCH\_11. The function 600 proceeds to Fig. 7 where, at step 600A, the function 600 calls itself recursively, this time passing to itself as the incoming argument CURRENT the name of the



patch PATCH\_10 which is the immediate predecessor (or CHILD) of the root patch named PATCH\_11, as can be seen in the patch tree 1504. The recursive function call begins again at the step 602 with the value CURRENT equal to the patch name PATCH\_10, and it proceeds again to Fig. 7, step 600A, where the subroutine 600 again calls upon itself recursively, this time to evaluate the next predecessor (or CHILD) patch named PATCH\_7. Again the function 600 commences at step 602 with CURRENT equal to PATCH\_7 this time, and program control proceeds again to Fig. 7, step 600A, where the same subroutine 600 is now recursively called twice during two successive passes through the loop defined by the series of steps 620, 600A, 622, and 900. During each pass through this loop, a different predecessor (or CHILD) patch of the patch named PATCH\_7 in the patch tree 1504 is evaluated. Two passes are required because there are two predecessor patches, one named PATCH\_6, and another named PATCH\_9. And in a like manner, when the function 600 is recursively called upon with CURRENT set equal to the name PATCH\_6, program control again proceeds to Fig. 7, step 600A, and the function again calls itself recursively twice to evaluate the two predecessor (or CHILD) patches in the search tree 1504 relative to the patch named PATCH\_6 – the patches PATCH\_5 and PATCH\_8.

[00081] In brief summary, it can be seen that each of the patches whose name appears in the patch tree 1504 is individually evaluated, and each such evaluation involves a recursive call to the function 600 with the CURRENT patch set to the name of the particular patch that is being evaluated during this call to the function. During these calls, the ratings of the various predecessor patches contained in the triples returned from the recursive calls, are studied and compared by further recursive calls to the rating of the CURRENT patch, and decisions are made as to which should be the recommended patches to present in the list of triples 1606 (Fig. 16) that is ultimately returned by the main calling function 500 to the step 108 in Fig. 1.

[00082] Having thus described an example of how the functions 500 and 600 operate upon specific data, and having explained the recursive nature of the function 600 and what it does, it now remains only to describe the details of the function 600, as presented in Figs. 6-9, during any one of these recursive

executions. In the paragraphs that follow, the function 600 is presumed to have been called upon, either by itself or by the function 500, to study specifically a patch whose name appears in CURRENT and its predecessor (or CHILD) patches in a patch tree or sub-tree. This study is conducted with due regard to the previously-installed patches whose names are included in the set variable INSTALLED, and this study focuses upon the patch tree whose root patch's name is contained in the variable ROOT.

[00083] Beginning at step 604, a test is made to see if the patch whose name appears in CURRENT has already been installed and thus appears in the array of patch names INSTALLED. If so, then there is no point in examining any predecessor (or CHILD) patches, since the system has already been updated beyond those predecessor patches. Accordingly, program control continues at step 606 where the single triple value CURRENT, NULL, ROOT is returned to the calling program. This says to the calling program that the patch name CURRENT is an installed patch, that there is no recommended replacement patch, and that the program which called the routine 600 should proceed with that as its only information concerning the remainder of the patch tree or sub-tree to the left of the patch CURRENT.

[00084] Assuming that the patch whose name appears in CURRENT has not been installed, then the function 600 proceeds to evaluate any predecessor (or CHILD) patches relative to the CURRENT patch. First, at step 608, the function 600 accesses the patch tree database 400 shown in Fig. 4, finds the patch tree having the root patch name that is stored in ROOT, searches the patch tree for the patch whose name appears in CURRENT, and then searches further to the left into the branches of the patch tree to find whatever number of immediate predecessor (or CHILD) patches may exist for the patch CURRENT. This set may contain no patches, one patch, or several patches. For example, the patch tree 1504 shown in Fig. 15 reveals that the patch named PATCH\_9 has no predecessor (or CHILD) patches. If PATCH\_9 is the CURRENT patch, the local set variable CHILDREN is set equal to a NULL value at step 608. On the other hand, the patch named PATCH\_10 has one predecessor (or CHILD) patch, the patch that is named PATCH\_7. Thus, if PATCH\_10 is the CURRENT patch, the set variable

CHILDREN is set equal to the single name PATCH\_7. But if the CURRENT patch is the patch named PATCH\_7, it can be seen that this patch has two predecessor (or CHILD) patches, the patches PATCH\_6 and PATCH\_9. Accordingly, if PATCH\_7 is the CURRENT patch, the set variable CHILDREN would contain only the two patch names PATCH\_6 and PATCH\_9.

[00085] Next, at step 618, four variables also local to each recursion of the function 600 are initialized. A set variable CHILDREN\_RESULT, which is used to recollect and store the triples (see step 108 in Fig. 1) returned by recursive function calls to the function 600, is initialized to the value NULL to signify that no triples have yet been found. Following each recursive call to the subroutine 600, any new triple values found are added to this set CHILDREN\_RESULT.

[00086] Another function variable CURRENT\_IS\_BETTER is initially set to the Boolean value FALSE. This is a flag which determines whether the patch whose name is in CURRENT is the best and recommended choice for installation, such that it should be recommended in lieu of any predecessor (or CHILD) patches (to the left of the patch CURRENT in the patch sub-tree starting with the patch CURRENT) in all of the triples that are returned by this particular recursive call to the function 600. That is what happens if, after the function 600 nears completion of its run, and has completed all of its recursive calls to itself, this flag is found to be set TRUE. On the other hand, if after analyzing recursively all of the predecessor (or CHILD) patches, the flag CURRENT\_IS\_BETTER is still found to be set FALSE, that means there are no patches which are predecessor (or CHILD) patches with respect to the patch CURRENT that are worse candidates for installation than the patch CURRENT. In that case, all of the triples that result from further recursive calls of the function 600 to itself to analyze the predecessor (or CHILD) patches are preserved and are simply passed back as return arguments from this particular recursion of the function 600, as will be seen.

[00087] Another function variable CURRENT\_SUPERSEDES\_INSTALLED is initially set to the Boolean value FALSE. This is a flag which will be set to TRUE if any triple returned from any recursive call to the function 600 for any predecessor (or CHILD) of the patch CURRENT contains the name of a patch in the installed component of the triple. This flag will have a value of TRUE if any of the

predecessors of CURRENT are in the set of INSTALLED patches. A value of TRUE will indicate that the CURRENT patch can only be recommended if it has a rating of 3 or a rating greater than the rating of at least one of the installed predecessors.

[00088] Another function variable

CURRENT\_IS\_BETTER\_THAN\_NEW\_REC is initially set to the Boolean value FALSE. This is a flag which will be set to TRUE if any triple returned by any recursive call to the function 600 for any predecessor (or CHILD) of the patch CURRENT, contains NULL for the installed patch and a recommended patch who's rating is less than or equal to the rating of CURRENT. If the value of CURRENT\_SUPERSEDES\_INSTALLED is FALSE and the value of CURRENT\_IS\_BETTER\_THAN\_NEW\_REC is TRUE then CURRENT becomes the patch recommended for installation used during the creation of the returned triples.

[00089] Continuing with the detailed description of the function 600, Fig. 7 describes the looping portion of the function 600, which recursively calls the function 600 itself (step 600A) to evaluate each and every predecessor (or CHILD) patch of the CURRENT patch, as well as the predecessors of those predecessor patches out to the ends of the patch trees. At step 620, a predecessor (or CHILD) patch is selected from the predecessor set CHILDREN. Its name is assigned to the variable CHILD. At step 600A, the function 600 is called recursively, and this time the CURRENT patch, the first argument passed to the function 600 called recursively, is the patch CHILD that was just selected. The values ROOT and INSTALLED remain unchanged and are passed to all of the recursive calls to the function 600.

[00090] The recursively called function may return 0, 1, or several triples of the kind described at step 108 in Fig. 1. These are collected and are stored as the value of the set variable CHILD\_TRIPLES at step 622. Next, the step 900, the details of which are shown in Fig. 9, begins examining each of the triples returned by the recursive call of the function 600. This examination, briefly summarized, searches for a triple with a non-NULL installed value indicating the flag CURRENT\_SUPERSEDES\_INSTALLED should be set to the value TRUE.

[00091] Additionally triples with Non-NULL installed patches are examined to determine if CURRENT would be a better recommendation than the patch currently recommended in the triple. If the triple contains no recommendation, determine if CURRENT is a good recommendation for the installed patch in the triple. Only one such triple needs to be identified to warrant setting the flag CURRENT\_IS\_BETTER to TRUE.

[00092] Additionally triples with no installed patch specified which contain a recommended patch are examined to determine if CURRENT is a better recommendation than the recommendation in the triple. If such a triple is found the value of CURRENT\_IS\_BETTER\_THAN\_NEW\_REC is set to TRUE.

[00093] Briefly summarized, this setting of the CURRENT\_IS\_BETTER flag causes all the triples generated by this particular operation of the function 600 to recommend the installation of the CURRENT patch, rather than some predecessor patch. In addition, once the CURRENT\_IS\_BETTER flag is set true, the checking process carried about by the step 900 is no longer needed and is essentially terminated for subsequent loops through the steps 620, 600A, 622, and 900 in Fig. 7.

[00094] When all of the predecessor (or CHILD) patches have been checked in Fig. 7, program control moves on to Fig. 8 where some final processing steps are carried out before operation of the function 600 terminates.

[00095] First at step 624, if no predecessor (or child) patch has been found to be installed and therefore the value of CURRENT\_SUPERSEDES\_INSTALLED is FALSE and the rating of CURRENT is greater than or equal to the rating of at least one recommended patch appearing in a triple resulting from a recursive call to function 600 (and therefore the value of CURRENT\_IS\_BETTER\_THAN\_NEW\_REC is TRUE), then the flag CURRENT\_IS\_BETTER is set equal to TRUE.

[00096] Next, at step 625, if no predecessor (or CHILD) patches have been found, then the CURRENT patch is selected as a RECOMMENDED patch if its ranking is 2 or greater. The flag CURRENT\_IS\_BETTER is set equal to TRUE, and this causes program control to move quickly through the steps 626, 636, 638 and 640. Nothing happens at 636, since there are no triples. At 638, a single triple

value recommending the installation of the CURRENT patch is generated, and at step 640, this single triple result is returned to the calling program.

[00097] The CURRENT\_IS\_BETTER flag is examined at step 626. If that flag is still FALSE, then program control normally moves rapidly through the step 628 to the step 634 where the set of triples CHILDREN\_RESULT is returned as a return argument from this execution of the function 600. Steps 628 and 630 check for the exceptional condition when there are no predecessor (or CHILD) patches (step 628) and the CURRENT patch is also the ROOT patch of the patch tree. In this one special case, at step 632, the triple (NULL, NULL, ROOT) is returned by the function 600. For example, this is the triple 1618 (Fig. 16) which results from the examination of the single element patch tree 1510 shown in Fig. 15. In this case no recommendation is made, since the PATCH\_16 has an unsatisfactory ranking of 1. Note that had the root patch had a ranking of 2 or greater, step 625 in Fig. 8 intervenes and causes the value (NULL, CURRENT, ROOT) generated at step 638 to be returned. This is illustrated by the exemplary triple 1616 shown in Fig. 16 that corresponds to the trivial patch tree 1508 shown in Fig. 15, where the single patch PATCH\_15 has a ranking of 2.

[00098] Returning to the step 626, if the flag CURRENT\_IS\_BETTER has been set TRUE, then at step 636, all of the returned triples are examined, and those triples that do not name a predecessor patch are discarded. The remaining triples are transferred to a new set variable called RESULT. In addition, these remaining triples are edited such that whatever recommendation they may have made is discarded and is replaced with the patch name stored as the value CURRENT such that no patch predecessor to the CURRENT patch is recommended. Next, at step 638, if all the triples are discarded and none remain, a single new triple is added to the set variable RESULT having the value (NULL, CURRENT, ROOT). In every case, the triples in the set named RESULT are then returned at Step 640.

[00099] With reference to Fig. 9, the subroutine 900 is shown which examines each of the triples in the set CHILD\_TRIPLES returned by a recursive function calls to the function 600 (step 600A in Fig. 7). At step 902, a triple is selected from the set CHILD\_TRIPLES. At step 904, this triple is added to the set CHILDREN\_RESULT which accumulates all of the triples generated during all of

the recursive calls to the function 600 made during this particular operation of an instance of the function 600. The remaining eight steps 908-922 performed by the subroutine 900 only need to be carried out until a recommended or existing patch is found that is inferior to the CURRENT patch, as indicated by the CURRENT\_IS\_BETTER flag having been assigned the value TRUE. Accordingly, at step 906, if that flag is set to TRUE, then the remaining steps in the subroutine 900 are skipped, and program control returns immediately to the step 902 where the next triple is retrieved and examined, and this process continues until all of the triples have been examined and added to the set CHILDREN\_RESULT by the step 904.

[000100] Assuming that the flag CURRENT\_IS\_BETTER is still false, for each triple, program control continues at step 908 where the triple is examined to see if it contains an installed patch. If it does not, then at step 910 the rating of the triple's recommended patch (if one exists) is compared to that of the CURRENT patch. If the CURRENT patch's rating is greater than or equal to that of the recommended patch, then at step 912 the flag CURRENT\_IS\_BETTER\_THAN\_NEW\_REC is set equal to TRUE. Otherwise, the flag is not adjusted and in either case program control returns to step 902.

[000101] Back at step 908, if the triple did contain an installed patch, then at step 914 the CURRENT\_SUPERSEDES\_INSTALLED flag is set equal to TRUE. Then at step 916 the triple is examined to see if it contains a recommended patch. If it does, then at step 918 the rating of the triple's recommended patch is compared to the rating of the CURRENT patch. If the CURRENT patch's rating is greater than or equal to the rating of the recommended patch, then at step 922 the flag CURRENT\_IS\_BETTER is set equal to TRUE. Otherwise, the flag is not adjusted and in either case, program control returns to step 902 where the next triple is examined.

[000102] Back at step 916, if the triple did not contain a recommended patch, then at step 920, the rating of the CURRENT patch is examined. If it is equal to 3, then at step 922, the CURRENT\_IS\_BETTER flag is set equal to TRUE. Likewise if the CURRENT patch's rating is greater than the rating of the installed patch specified in the triple under examination, then again, at step 922, the flag

CURRENT\_IS\_BETTER is set equal to TRUE. Otherwise the flag is not adjusted and in all cases, program control returns to step 902 where the next triple is examined.

[000103] This looping process in Fig. 900 continues until all of the triples have been examined and added to the set CHILDREN\_RESULT so that all of the triples can optionally be examined and altered by the code shown in Fig. 8 (described above) after the function 600 stops calling the subroutine 900.

[000104] With reference to Fig. 17, a second embodiment of the present invention is presented which introduces several additional strategies into the process of searching for a suitable set of patches. In this variation, the patch chains are searched in the forward direction, from a particular patch that is known to correct a particular defect or to introduce a particular property forward through the patch chain to its root (from left to right in the figures, rather than from right to left as taught above), checking each patch for its suitability for a particular user, given the degree of reliability that is indicated by the nature of the user's application.

[000105] In addition to the patch reliability rating, shown in parenthesis in Figs. 4, 13, 14, and 15 and discussed fully above, two other factors may be assigned to each patch to improve and to add flexibility to the patch selection process.

[000106] The first additional factor is patch visibility -- whether a patch is visible to "All" users, to a "Limited" number of users, or to "None." The README file that accompanies a patch with a visibility of "None" cannot be found and cannot be searched by users searching for patches, and thus these patches are entirely out of service and are not even visible. The README file that accompanies a patch with a visibility of "Limited" can only be viewed and searched by specially empowered users. Accordingly, patches whose visibility is limited simply do not exist insofar as other non-empowered users are concerned -- such users cannot even search the patch properties to see what defects they cure, since searches of all the patch README files skip over the README files for such patches.

[000107] The second additional factor is patch availability. A patch may be available to "All," or its availability may be "Limited" so that it may be accessed



(or downloaded) only by especially empowered users, or "None" may access the patch. Such a patch is said to be "Archived," since it can be found, but it cannot be downloaded and installed into a system.

[000108] Each user of the patch system is considered to be fulfilling a particular role. Users follow a variety of use models, have different needs, and their authorization to access patch information is also different. For example:

[000109] An "external user" is considered to be a system administrator representing a particular company. The external user will be able to view and search for patches which have a visibility attribute of "All" and may download those patches which have an availability attribute of "All." When presenting representative patches from located patch chains to this type of user, it is desirable to make a conservative recommendation based upon the patch ratings as well as to show the latest version on the chain. But some external users may ask to receive, in addition, a less conservative but more current recommendation.

[000110] An "internal user" is considered to be an internal patch expert, locating patches on behalf of an external client. The internal patch expert is allowed to search and view patches which have "Limited" visibility or availability. Again, when displaying representative patches from located patch chains, a conservative recommendation is made, but a less conservative recommendation of the latest acceptable patches may also be made. The list of patches which such an "internal user" is permitted to view may also differ from an external user's view as a result of the internal user's expanded authorizations. It is thus possible to define a wide variety of real and automated "users" of the system that are assigned different combinations of rating, visibility and availability values.

[000111] In general, all user types have a common overall use model:

[000112] Identify a patch or patches that solve a particular problem;

[000113] Analyze those and subsequent patches (those closer to the root of the same patch tree); and

[000114] Obtain patches that are suitable to the requirements and permissions of each given user.

[000115] However, during the identification process, each type of user will have unique expectations for patch recommendations.

[000116] In a typical case, a user, having a system built from specific hardware and running a specific operating system, is seeking a specific patch (or patch set) to fix a particular defect, as was explained above. Once the appropriate patch (or patch set) is identified, the user downloads the patch (or patches) and installs it (or them) on their system.

[000117] The user begins the process, after identifying a universe of patches intended for use with particular programs and/or a particular operating system, by running some form of patch search. If the user has some key words that identify the problem (such as "sendmail" core dumps, for example), the "defect" descriptions or "README" files for all of the patches can be searched 1702 (Fig. 7) in the hopes that a suitable patch to fix the defect can be found. If a particular patch (for example, "PATCH\_7" within patch tree 1504 in Fig. 15) is found whose README file matches the pattern of key words searched for, it is then checked to see what its displayability rating is ("ALL", "LIMITED", "NONE") and to then determine whether it can be displayed to this particular user of the system (step 1706). In this manner, one or more displayable patches relevant to the user's needs are identified at step 1708. Next, using these patches as a starting point, their respective patch trees 1504 are found and are searched towards the root of each such patch tree. For example, if "PATCH\_7" in the tree 1504 (Fig. 15) is found, the patches PATCH\_10 and PATCH\_11, which have the same desired properties as PATCH\_7, are ALSO examined. These are more current patches than PATCH\_7, probably having additional desirable properties, but they may be less reliable than PATCH\_7. If the user is seeking a patch to fix a specific program defect that is defined by a pattern of key words found in PATCH\_7's README file, PATCH\_7 is the first of a series of patches PATCH\_7, PATCH\_10, and PATCH\_11 in the patch tree 1504 all of which can fix this same defect.

[000118] Alternatively, the user may begin seeking a patch by searching for words and phrases in one or more other external databases, such as a sendmail web site database (step 1704), and may possibly obtain from such a site a message suggesting the id of a patch that fixes a particular problem. The process of finding,

selecting, and examining patches then proceeds through the steps 1706, 1708, and 1710 as was just described above.

[000119] Each of the patches is now scanned, using programs such as those shown in the Appendix suitable to the needs and desires and classifications of each given user, starting with the patch PATCH\_7 and continuing to the right in Fig. 15 to the root of the patch tree 1504, or vice versa. Thus, the patches PATCH\_7(2), PATCH\_10(2), and PATCH\_11(1) in the patch tree 1504 are checked. The programs that carry out this checking are similar to those already described above, except that these programs scan patches in a direction leading towards the root of a patch tree, rather than away from the tree root towards the branches of the trees. Accordingly, the tree search algorithm is simplified, since multiple branches do not need to be scanned, and not all the patches in a given patch tree need to be checked – only the more recent patches towards the root of the patch tree from the starting point of this search.

[000120] As this search is carried out, step 1712 in Fig. 17, the availability rating of all patches retrieved should comply with or surpass the availability rating assigned to or selected by a particular user or recipient, as indicated at step 1714; and the reliability rating of all patches must should match or exceed the reliability rating appropriate to or selected by the given user, in accordance with that user's need for reliability (step 1716).

[000121] As a result of such a patch search for patches that cure specific defects or have specific properties, a specific set of patches PATCH\_7, PATCH\_10, and PATCH\_11 within the patch tree 1504 are identified each of which patches can cure a particular defect or provide a specific property. These must also be visible to the user, available to the user, and satisfactory from a ratings point of view to the user.

[000122] The user is next presented with representative patches selected from this specific set of patch chains at step 1718 in Fig. 18. In an embodiment of the invention, the user may be presented with patch recommendations organized into two, three, or more columns. For example, and as shown at 1720 in Fig. 18, the patches may be presented in three columns to a user who is not particularly concerned about risk, since the computer is the user's personal machine and is not

assigned any “mission critical” tasks. But if the user wished to avoid any significant risk, then the third column of patches would not be included. (See the programs in the Appendix for examples of different types of patch searches.)

[000123] While several embodiments of the invention have been described, it is to be understood that modifications and changes will occur to those skilled in the art to which the invention pertains. Accordingly, the claims appended to this specification are intended to define the invention precisely.

## COMPUTER PROGRAM LISTING APPENDIX

---

Identity -- Return a current patch on a chain:

```
public PatchDescription locate (PatchDescription patch)
{
    return patch;
}
```

---

RecommendedVisibleAvailable -- Find the last highest rated patch which is also visible and available. This is the recommended patch for external users. This is a conservative recommendation. The patch must be rated at least 2.

```
public PatchDescription locate (PatchDescription patch)
{
    int bestRating = 0;
    PatchDescription best = null;

    while (patch != null) {
        int rating = patch.getRating();

        if ((rating > 1) &&          // no bad patches or 1 rated patches
            (rating >= bestRating) &&
            patch.isVisible() &&    // no non visible patches
            patch.isAvailable() )  // no non available patches
        {
            bestRating = rating;
            best = patch;
        }
        patch = patch.getReplacement();
    }
    return(best);
}
```

---

LastVisibleAvailableGood -- Find the last patch on the chain which is both visible and available and good (has a rating of atleast 1).

```
public PatchDescription locate (PatchDescription patch)
{
    PatchDescription best = null;

    while (patch != null) {
        int rating = patch.getRating();
```

```

        if ((rating > 0) &&    // ignore bad patches
            patch.isVisible() && // ignore invisible patches
            patch.isAvailable() ) // ignore non available patches
        {
            best = patch;
        }

        patch = patch.getReplacement();
    }

    return(best);
}

```

---

Recommended -- Find the last highest rated patch on the chain.  
The patch must have a rating of at least 2.

```

public PatchDescription locate (PatchDescription patch)
{
    int bestRating = 0;
    PatchDescription best = null;

    while (patch != null) {
        int rating = patch.getRating();

        if ((rating > 1) &&    // no bad patches or 1 rated patches
            (rating >= bestRating) )
        {
            bestRating = rating;
            best = patch;
        }

        patch = patch.getReplacement();
    }

    return(best);
}

```

---

LastGood -- Find the last patch on the chain with a rating of  
atleast 1.

```

public PatchDescription locate (PatchDescription patch)
{
    PatchDescription best = null;

```

```

while (patch != null) {
    int rating = patch.getRating();

    if (rating > 0) {    // ignore bad patches
        best = patch;
    }

    patch = patch.getReplacement();
}

return(best);
}

```

---

Best -- Find the best patch on the chain. It will be the last highest rated patch on the chain. No minimum rating is specified.

```

public PatchDescription locate (PatchDescription patch)
{
    PatchDescription best = patch;
    int bestRating = best.getRating();

    while (patch != null) {
        int rating = patch.getRating();

        if (rating >= bestRating) {
            best = patch;
            bestRating = rating;
        }

        patch = patch.getReplacement();
    }

    return(best);
}

```

---

BestVisibleAvailable -- Find the best patch on the chain which is both visible and available. It will be the last highest rated patch on the chain. No minimum rating is specified.

```

public PatchDescription locate (PatchDescription patch)
{
    int bestRating = -100; // lower than any reasonable rating.
    PatchDescription best = null;

    while (patch != null) {

```

```
int rating = patch.getRating();

if (patch.isVisible() &&
    patch.isAvailable() &&
    (rating >= bestRating) )
{
    bestRating = rating;
    best = patch;
}

patch = patch.getReplacement();
}

return(best);
}
```